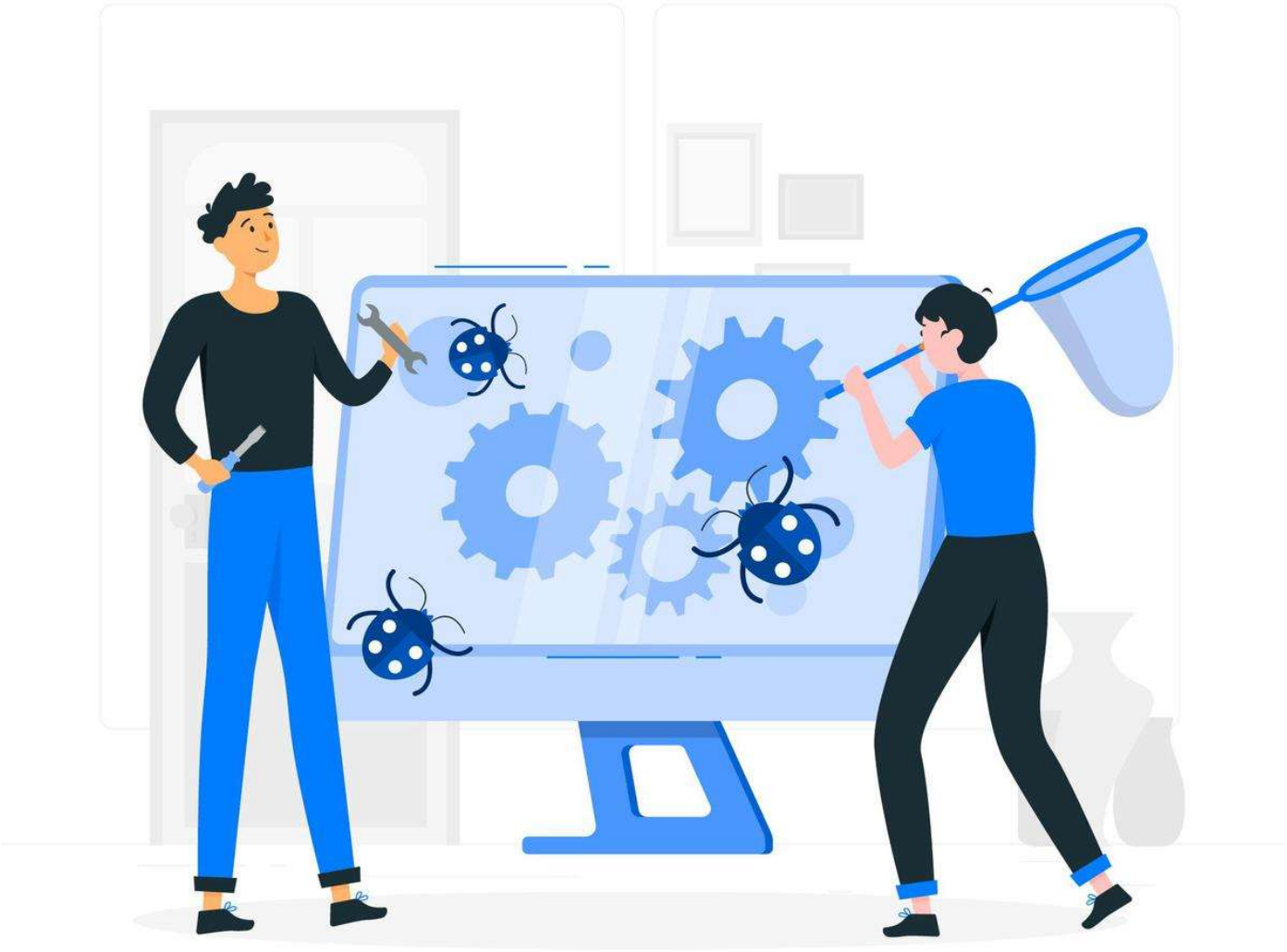
ARTÍCULOS DE TECNOLOGÍA

Depurando aplicaciones en el navegador

Priscila Storck

05/09/2022



¿Qué es un Bug?

Antes que empecemos a pensar en formas de entender y solucionar bugs es necesario que sepamos lo que es un bug. Un bug es un error en el diseño, desarrollo o ejecución de un programa que hace que tenga un comportamiento inesperado, diferente al planeado. Muchos son los factores que pueden generar un bug, entre ellos: un error de digitación, un problema de lógica o de importación, entre otras causas.

El console.log es uno de nuestros amigos

Es muy común qué al comenzar a programar uno utilice mucho el console.log para entender los bugs. Ahora imaginemos que estamos trabjando en una aplicación real que fue creada por otra persona y está llena de bugs. ¿Será que el console.log es la mejor herramienta para resolver estos errores lo más rápido posible?

La respuesta es, probablemente, no. Usando el console.log en esta situación posiblemente tendríamos que ponerlo manualmente en todas las funciones del programa y no llegaríamos a la solución de nuestros problemas, pues los valores impresos podrían ser inconclusivos y nos revelarían muy poco del flujo de la aplicación. Para estos casos, utilizar el debugger de Chrome es la mejor alternativa.

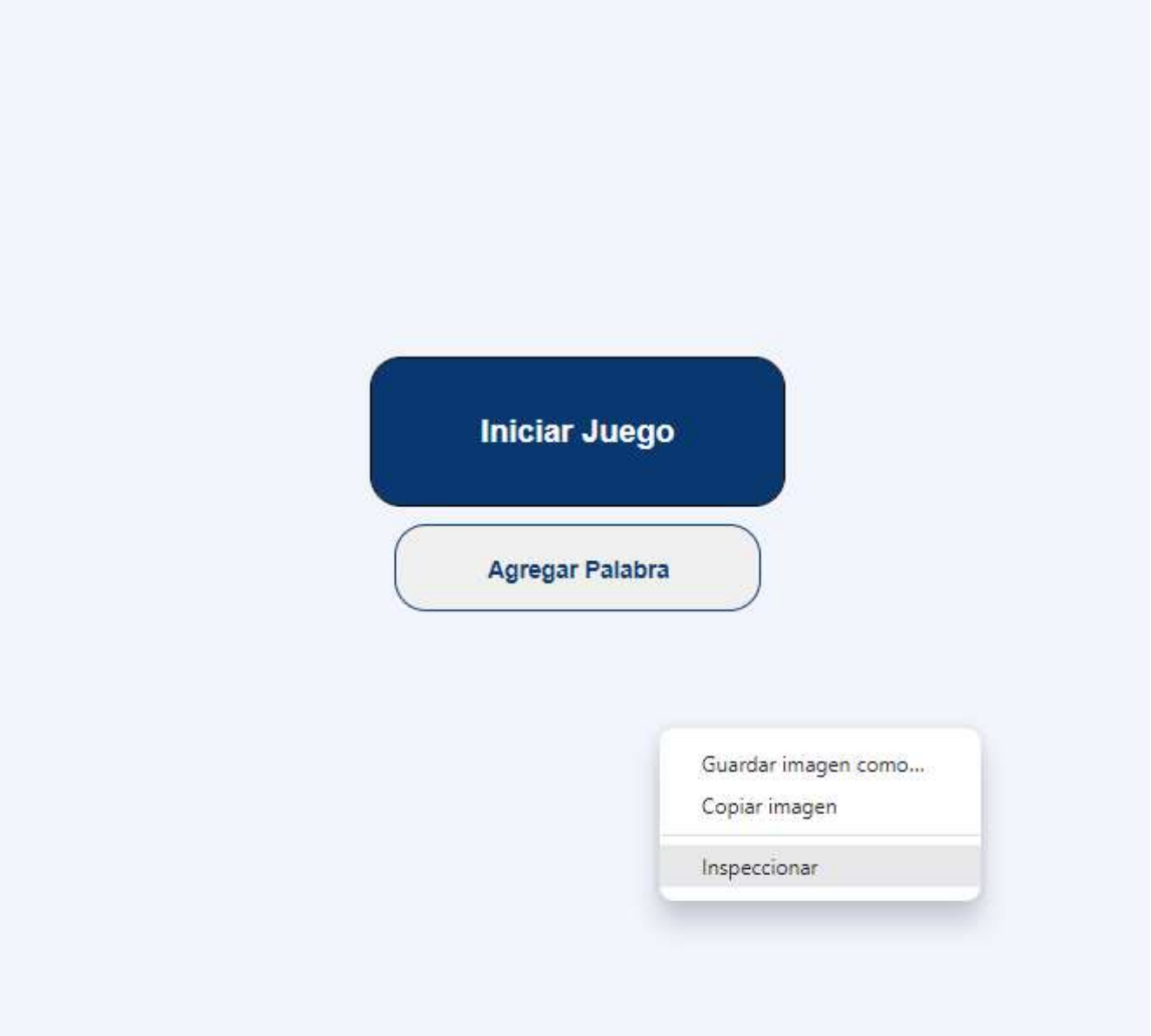
Debugger Javascript en el Google Chrome DevTools

Debug, debugging o en español depuración, por su vez, es el proceso de encontrar y remover un error en un programa. Hay muchas herramientas que nos permiten inspeccionar el código fuente de un programa. Por defecto, todos los IDE contienen este proceso de depuración.

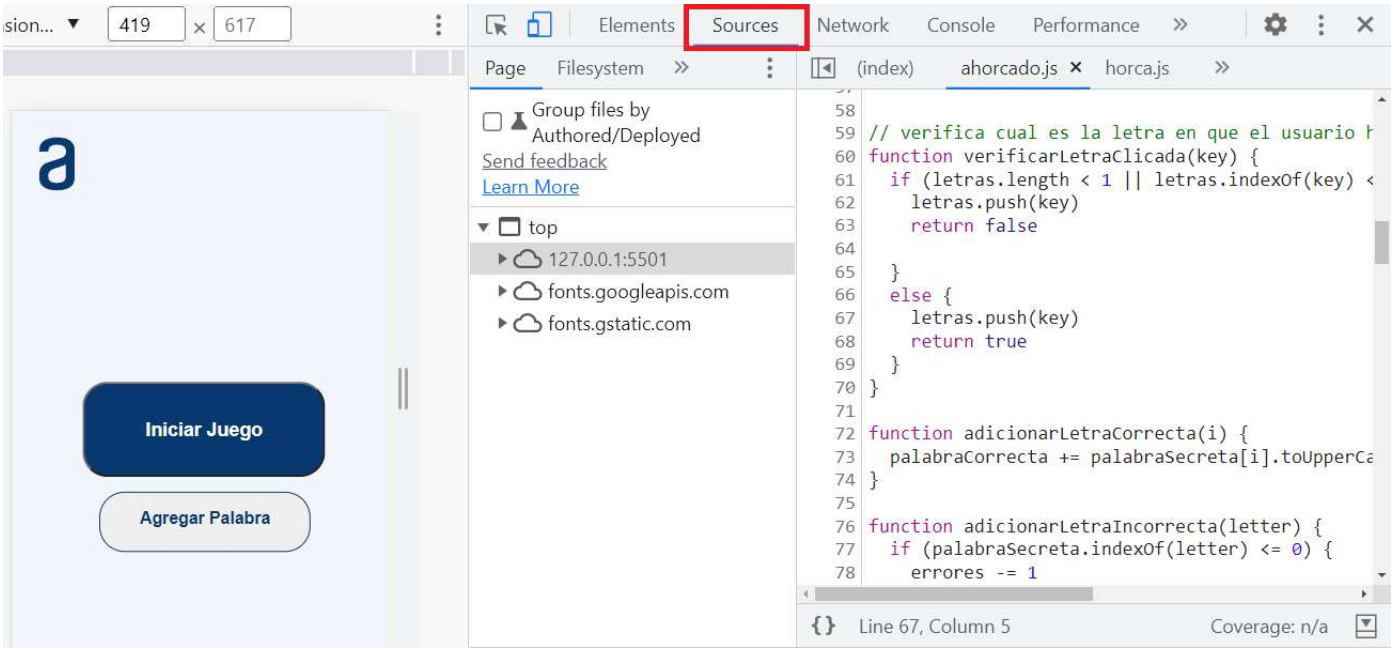
En este artículo vamos a enfocarnos en aprender cómo se hace un debug a través del navegador, empleando la herramienta de desarrollador de Google, también conocida como Chrome Developers Tools. Pues, es una herramienta que todo desarrollador web debe conocer por su simplicidad y practicidad.

¿Cómo utilizar el debugger en el navegador?

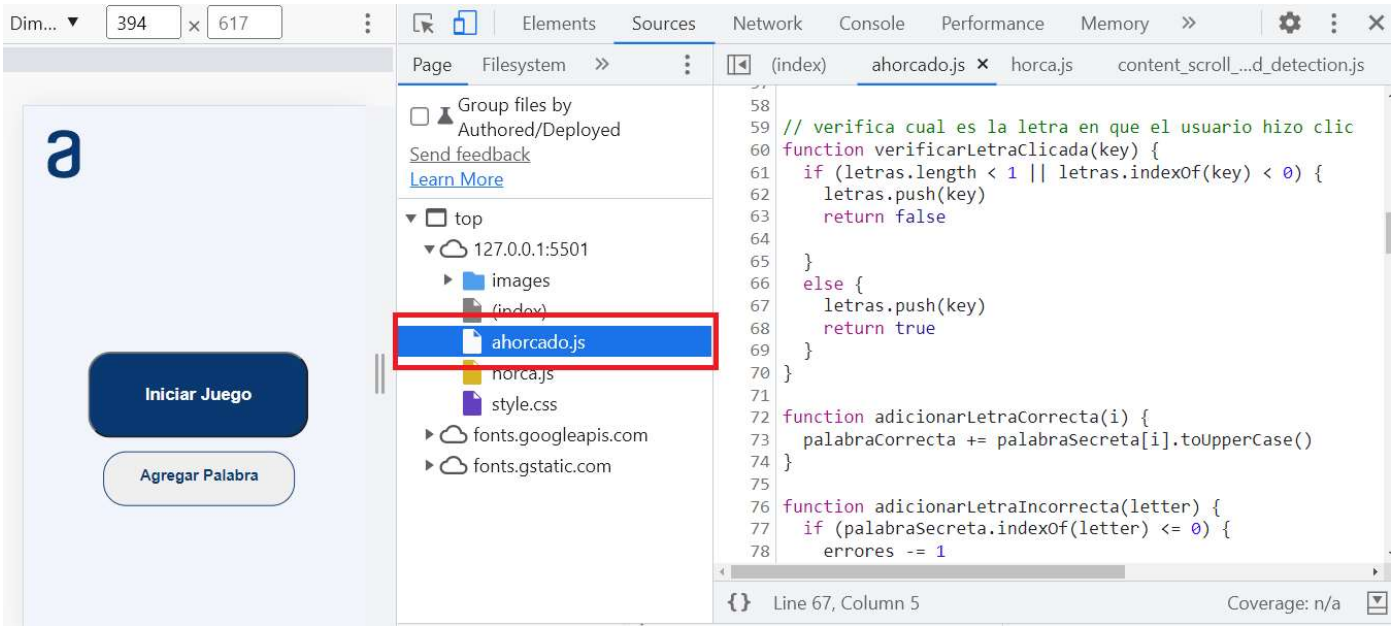
Cuando descubrimos un bug en nuestro programa, lo primero que debemos hacer es depurarlo haceiendo click con el botón derecho del ratón e **inspeccionar**.



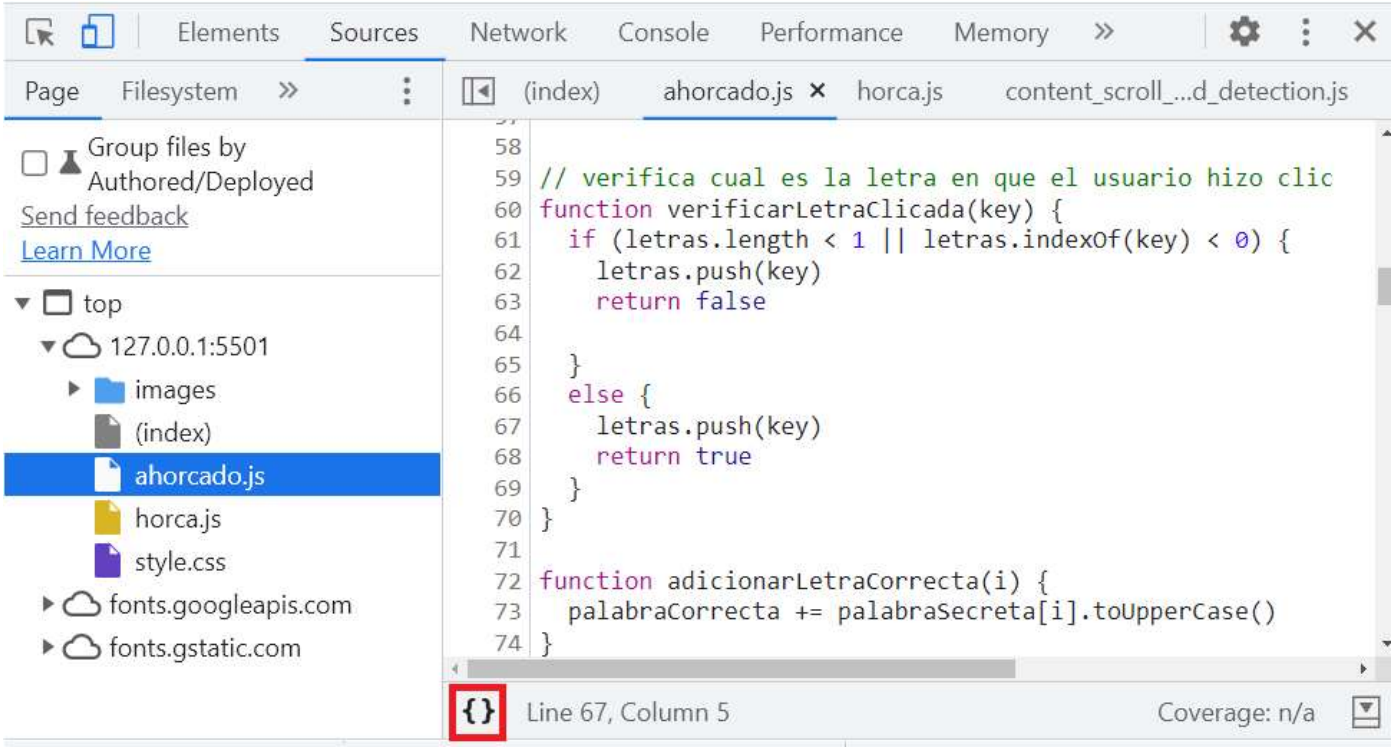
En el encabezado de la pestaña tenemos algunas opciones como Elements, Sources y Network. Hagamos click en la pestaña Sources.



En la pestaña Sources hay varias carpetas como en la imagen abajo. Esta organización refleja la estructura de directorios de nuestro programa. Tenemos que abrir las carpetas y buscar el archivo JavaScript que queremos analizar, es decir, si estamos intentando entender, por ejemplo, lo que está pasando en el archivo index.js, hay que buscarlo ahí con este mismo nombre.



Posiblemente el código estará sin formatación. Para formatearlo solo necesitamos hacer click en las llaves que están al final de la pestaña y el código aparece cómo en nuestro editor.



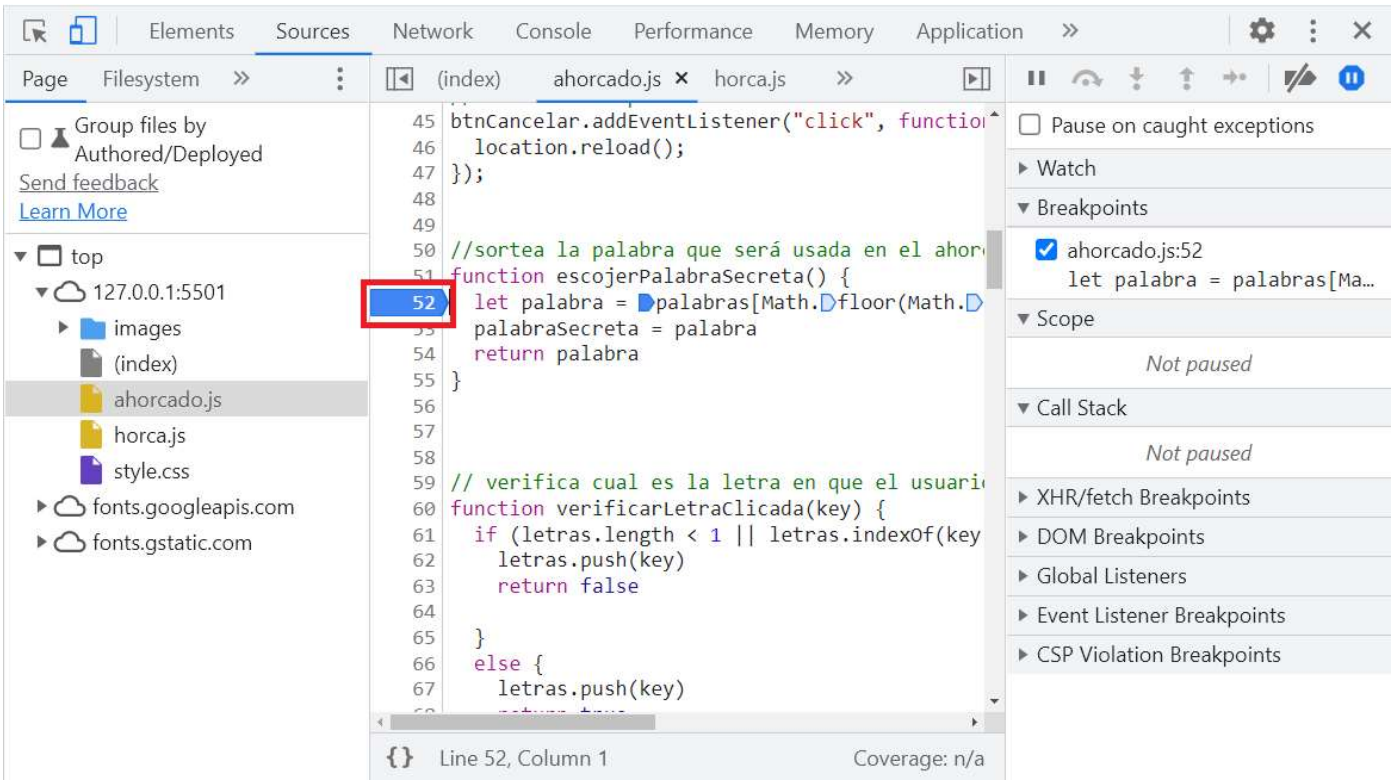
Breakpoints, depuración del código punto a punto.

Como vimos, en la pestaña Sources tenemos todos los recursos necesarios para debugar un archivo JavaScript. En primer lugar, vamos a conocer el recurso que es una de las piedras fundamentales en el proceso de investigación del código: el *breakpoint*.

Los *breakpoints* pausan la ejecución del programa en un punto específico del código para que, a partir de ahí, podamos analizar las informaciones que el programa está recibiendo.

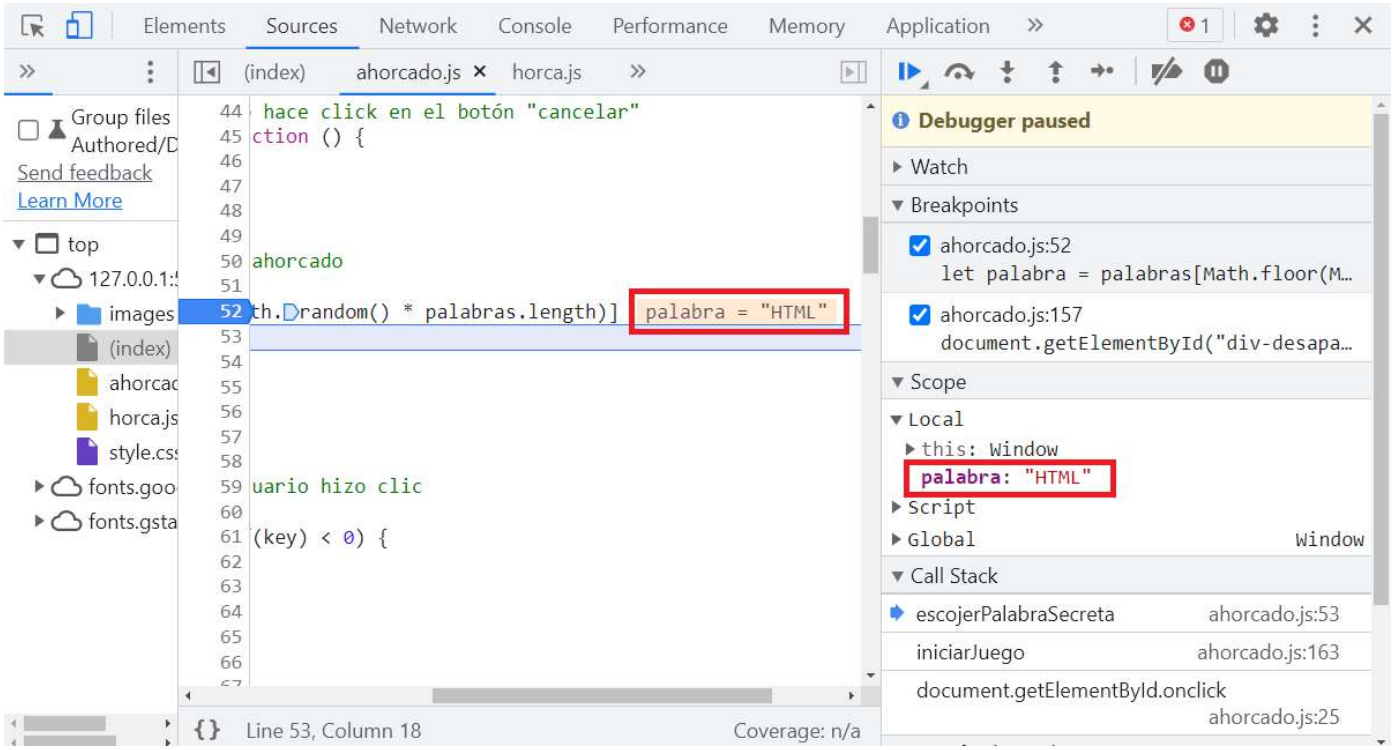
En el esquina derecha de la pantalla tenemos varios recursos de *breakpoints* para apoyarnos en la búsqueda de los bugs de nuestro código. Decidir en qué punto del código vamos a poner los *breakpoints* va a depender de lo que queremos investigar.

Entre los varios tipos existentes de *breakpoints* lo más utilizado es el marcado de línea. Solo necesitamos hacer clic en el número que aparece al lado izquierdo de la línea de código. ¡Así de simple! Por ahora, vamos a configurar un breakpoint de esa manera sencilla y específica.

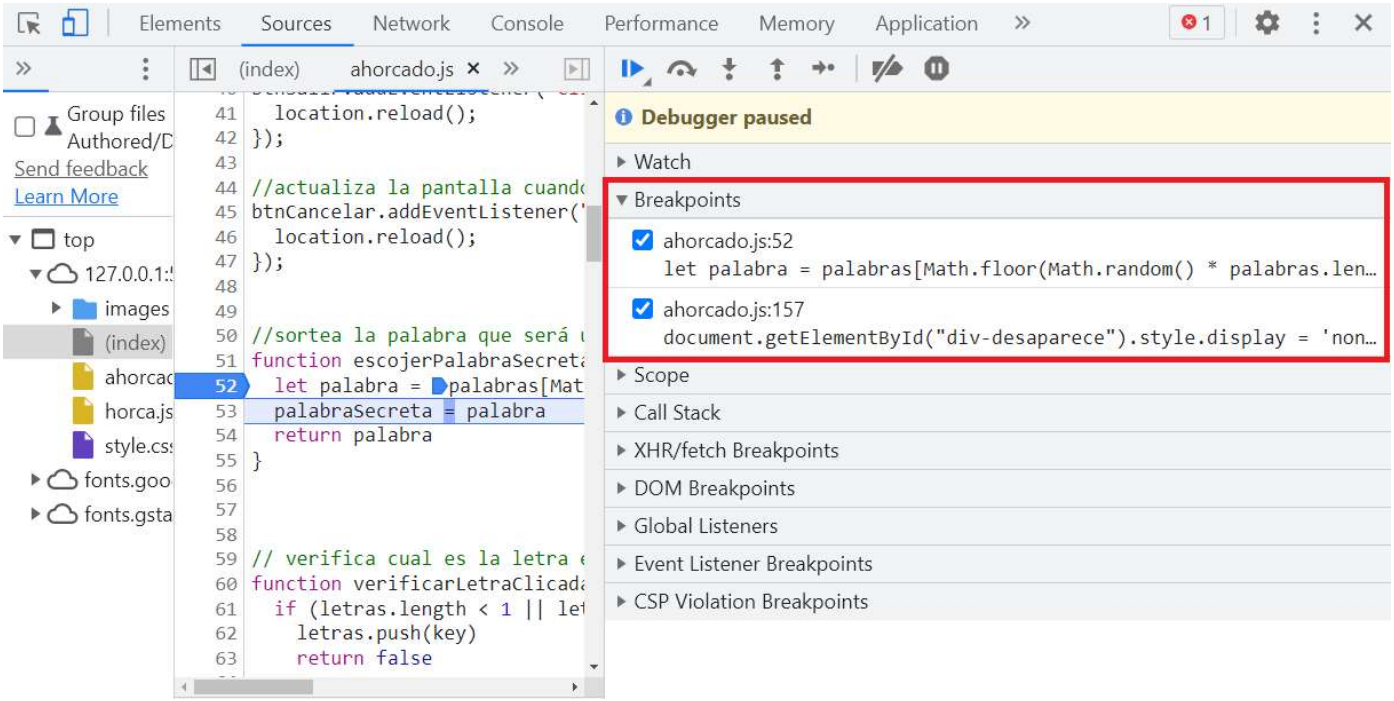


Elegimos la línea que queremos analizar, hacemos clic en ella y apretamos F5. Al recargar la página, empezamos a debugar el programa a partir del *breakpoint* que definimos

Verificamos que se ha detenido el proceso de ejecución del archivo JavaScript, en la línea que elegimos. Ahora los valores de las propiedades de las funciones son exhibidos en la pantalla, hasta donde pusimos el *breakpoint*.



Después de nuestra detención estratégica, vamos a explorar un poco de las herramientas de debug DevTools que se muestran en la esquina derecha de la pantalla. En la pestaña "*breakpoints*" son exhibidas todas las líneas de código en las cuales pusimos *breakpoints*. Eso nos permite tener un mejor control de nuestros debuggers, además de poder desactivarlos haciendo clic en el ícono ✓.

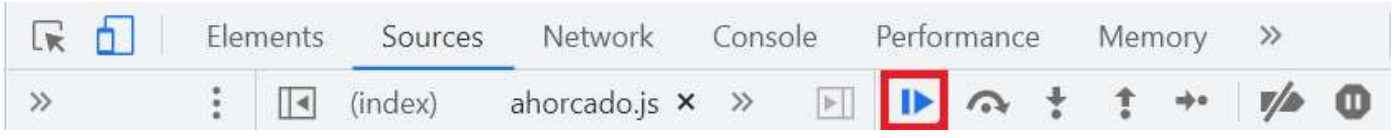


Comandos de breakpoint

Ahora vamos a echar un vistazo en la parte superior de la pestaña donde están los comandos de *breakpoint*.

En continuación tenemos los comandos, que nos permiten analizar el código de la forma más útil, dependiendo de la situación:

- **"Resume script execution"** ejecuta el código, hasta que encuentre un *breakpoint*.



- **“Step over”**: no entra a la función, sino que la ejecuta sin más detalles y luego salta para la función siguiente. Es utilizado en contextos en que varias funciones son llamadas en un mismo lugar, pero solo si quiere examinar una de ellas, saltando las restantes.

- **“Step into”**: al contrario del “Step over”, entra a la función y nos detalla todas las informaciones que están dentro. Se usa cuando sabemos que una función específica tiene un bug y queremos entender cuáles son los valores pasados ahí.

- **"Step out”**: al utilizarlo salimos de la función.

- **“Step”**: al hacer click en este ícono, vamos analizando el código línea por línea.

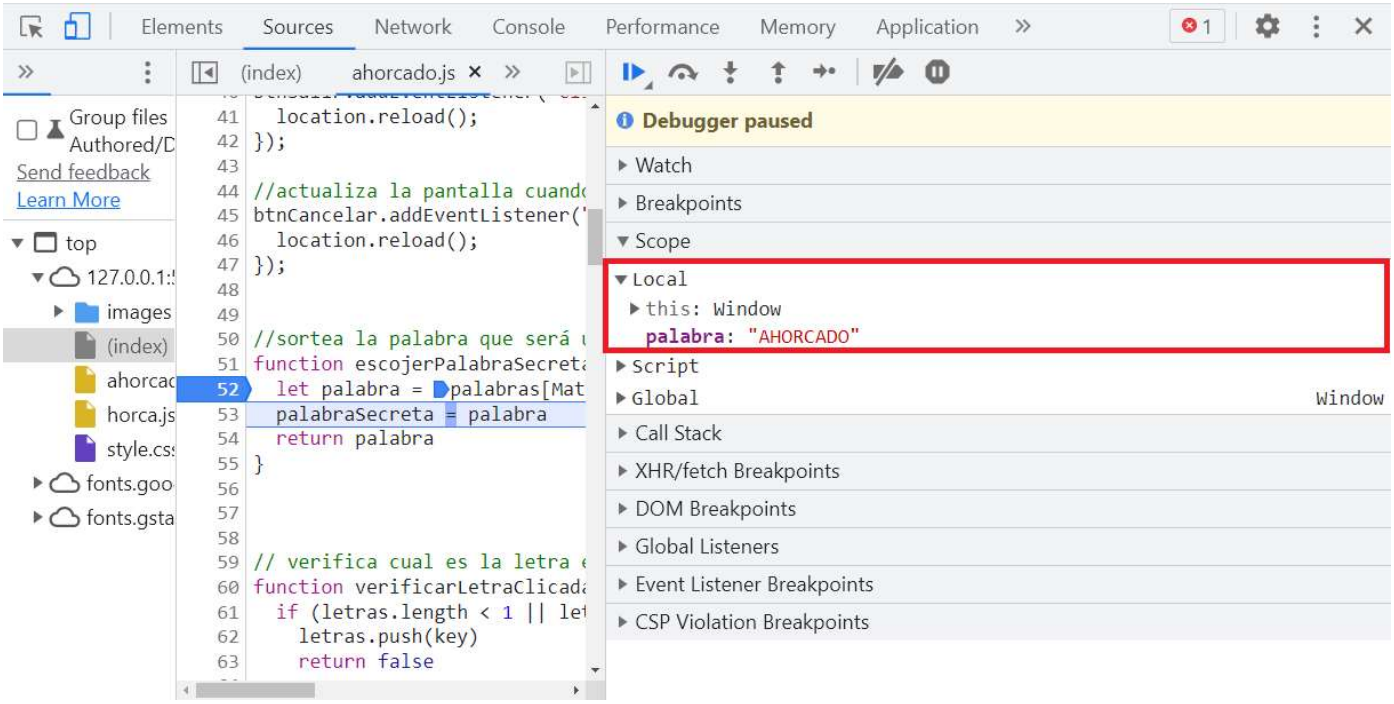
- **“Deactivate breakpoints"**: lo empleamos cuando necesitamos desactivar todos los breakpoints de una vez.

- **"Pause on exceptions”**: busca una excepción por todo el código y pone el punto para detenerse si encuentra alguna.

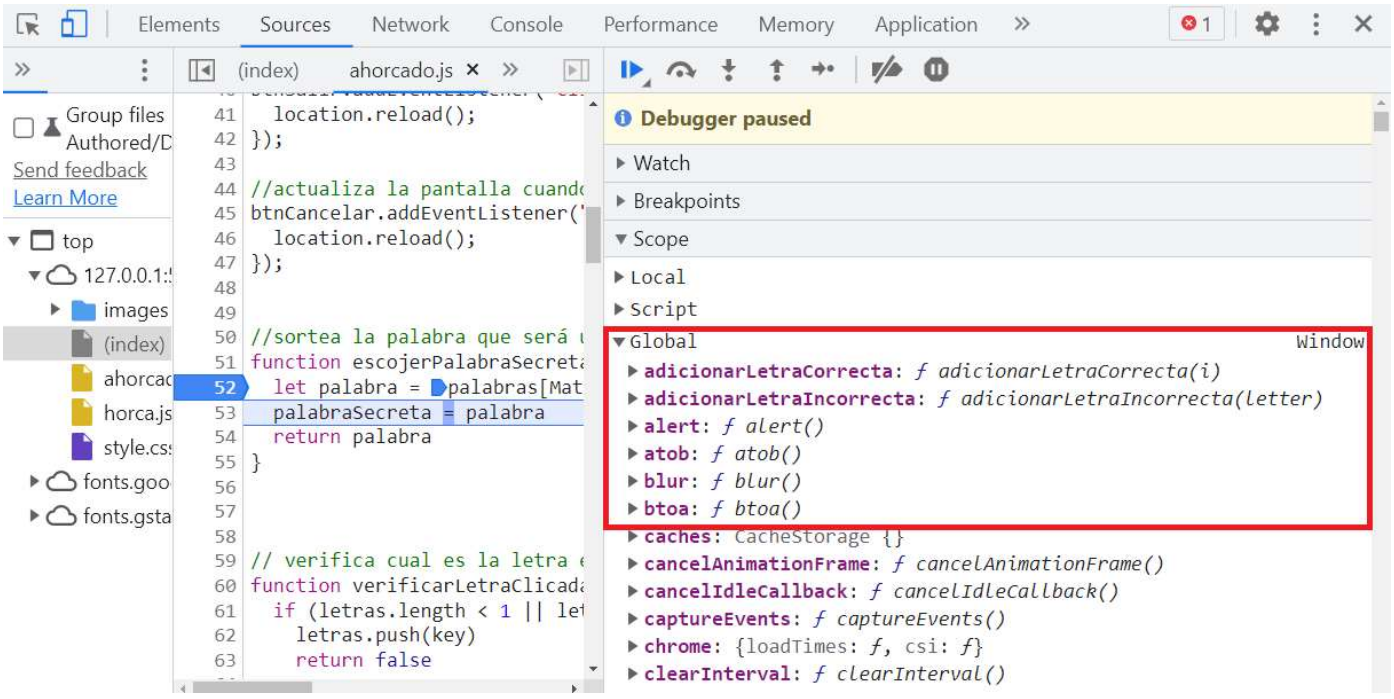
Scope

En la pestaña “Scope” están todas las informaciones de los ámbitos local y global.

En la parte “Local” aparecen las propiedades y variables que fueron declaradas en las funciones.



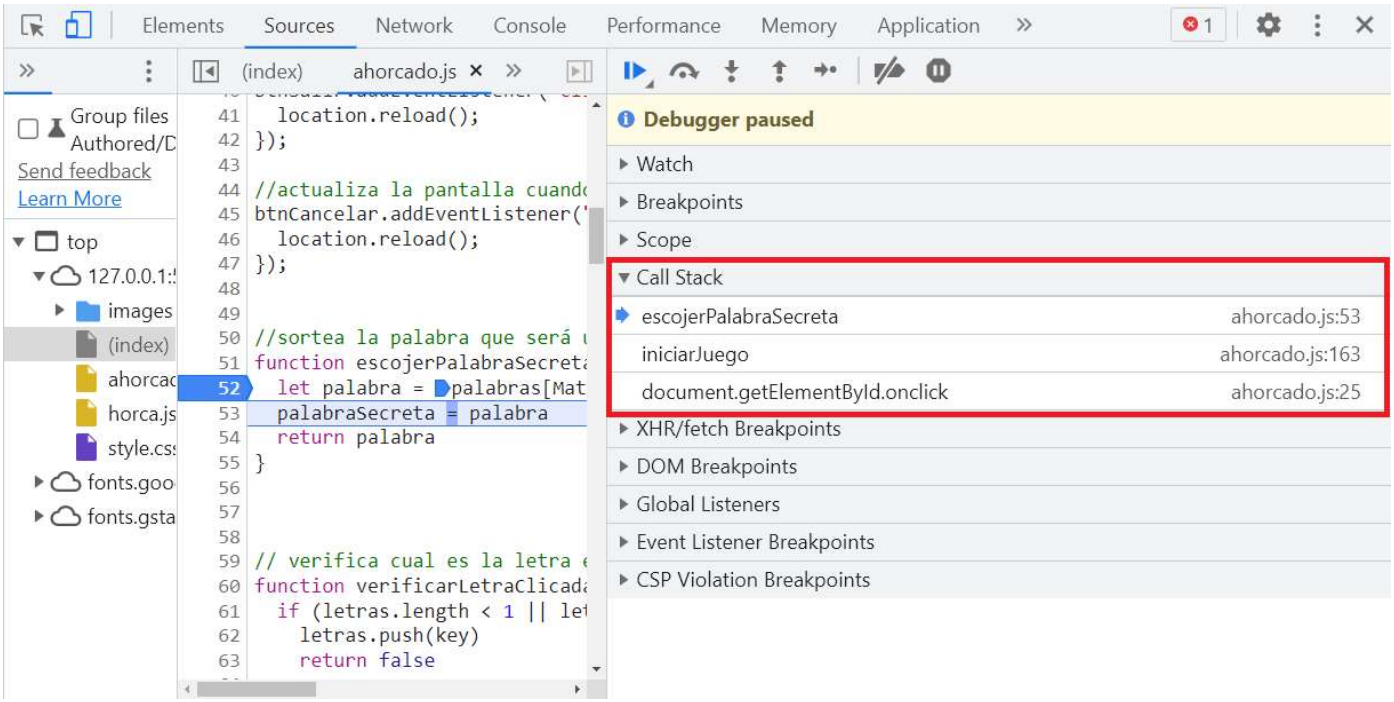
En el ámbito global están las variables y propiedades que hacen parte del objeto Window, como, por ejemplo, el *alert*, que es muy utilizado en aplicaciones web.



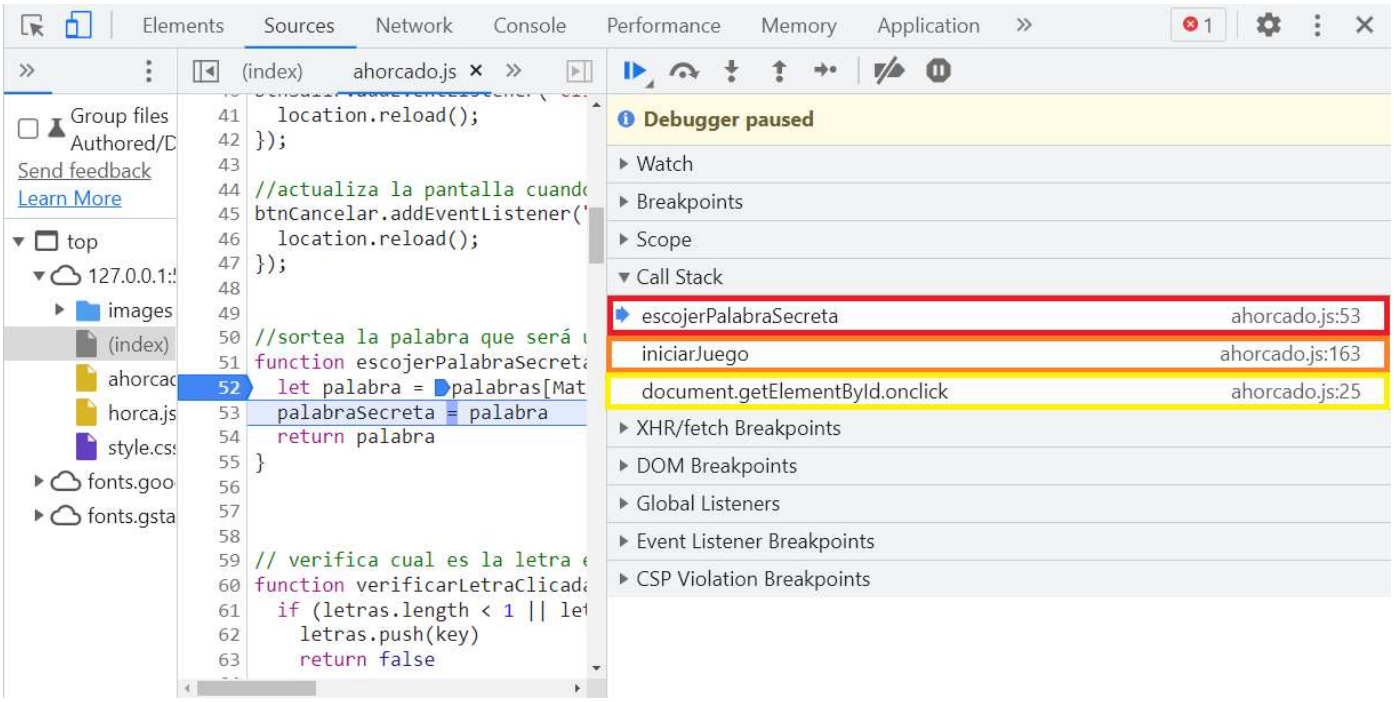
Call Stack

Otro recurso que también puede auxiliar nuestro análisis, es la pestaña "*Call Stack*" que nos muestra el camino hecho para que la función sea ejecutada.

Esta sección funciona como el historial del código, dentro de la misma podemos mirar cuál fue la función que la llamó y entender mejor el flujo del programa. Para ejemplificar esta situación, miremos la imagen en seguida:



En "*Call Stack*" vemos que actualmente nuestro *breakpoint* está dentro de la función `escojerPalabraSecreta()` señalada en rojo. Abajo de esta línea tenemos la función `iniciarJuego()`(naranja)que es la que llamó la función `escojerPalabraSecreta()` y, por fin, tenemos el evento `onclick`(amarillo)que llamó la función `iniciarJuego()` e inició todo el flujo del código.



CONCLUSIÓN

Podemos usar las herramientas nativas de Chrome para buscar errores en nuestras aplicaciones y entender el flujo de aplicaciones desarrolladas por otras personas. Esta forma de depuración hace que el trabajo del desarrollador sea más productivo y le ahorra tiempo para otras actividades.

Además de lo que aprendimos, hay mucho más información sobre la herramienta de depuración de Chrome. Para profundizar en el tema, te dejamos este enlace de la documentación oficial.

Hasta pronto y, ¡a debugar!

Sobre las autoras:

Barbara dos Santos es licenciada en Letras por USP y empezó a hacer su transición de carrera para el area de desarollo en 2020. Trabajó como desarrolladora front-end en consultoria y hoy actúa como instructora front-end en Alura Latam.

Priscila Storck estudia Análisis y Desarrollo de Sistemas, también ha hecho transición de carrera, trabaja como Scuba en Alura Latam y ve la tecnología como la herramienta de transformación más poderosa que tenemos hoy en día.

ARTÍCULOS DE TECNOLOGÍA